March 2010

Network Message Translation - The org.jprotocol Project

Gregory K. Holtorf
Worcester Polytechnic Institute

Project Number: GTH MQP-EZ09

The org.jprotocol Project

A Major Qualifying Project Report:
submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science
by

Gregory Holtorf

Date: March 8, 2010

Approved:

Professor George T. Heineman, Main Advisor

Problem Statement

The org.jprotocol project, sponsored by the Ezenia Corporation is designed to address incompatibilities in feature support over network communication between two or more clients in a client/server framework. To address the complications that come with increasing large application layer protocols and the need for backward compatibility, the jprotocol extension helps simplify and translate application communication.

Background

The Ezenia Corporation develops and sells collaboration software. They compete with open-source free software (such as OpenFire and ejabberd) as well as products from other development corporations, such as IBM Lotus Notes, AOL Instant Messenger and Google Talk. MxM offers a variety of collaborative features, including instant messaging, text chat, video conferencing, and whiteboard.

MxM is designed as a client/server architecture where all collaborative communication between clients is directed through the MxM server. Ezenia is attempting to validate a proof of concept port of various features within MxM to the iPhone [1]. During my internship at Ezenia, I developed a standalone iPhone application to support the text chat feature between the iPhone application and the MxM system. The goal of this Major Qualifying Project (MQP) is to investigate how to support the interoperability of multiple versions of clients and servers as the protocol and technology adapts and matures.

In particular, Ezenia was interested in extending their whiteboard system. This application prototype is a domain-specific instance of a more general problem, namely, creating a client/server framework where the clients (and possibly servers) each support different features (with possibly different implementations). This is an interesting problem because in the traditional model, it is expected that the client application is upgraded synchronously with the server based system to which it connects. In addition, in the traditional application domain, clients and servers do not "announce" which features they support, but rather only declare at run-time whether a request was satisfied or not, so there is no *a priori* ability to determine whether a client will be able to communicate with a different server version.

Methodology

During the summer of 2009 Michael Fitzell, Vice President of Engineering at Ezenia proposed continuing working on the project I was involved in as an MQP. I would attempt to create a whiteboard component to the iPhone MxM client that I built that summer. After professor Heinemann agreed to sponsor the project it was decided that the project would need three parts. The first would be a small iPhone client. The second would be a framework on the servers that would allow the existing clients to communicate with the iPhone and allow the whiteboard to be easily expanded to other platforms without further re-writing. The third element would be the academic element, namely a general implementation of the whiteboard framework that could be shared publicly.

During this project I ensured that I did not release any confidential information or code from Ezenia to the general public.

During A term, project time was devoted to research, writing a proposal, and getting everything approved. Research was conducted in multiple areas: other successful feature translation services, services that use a similar methodology to how we want to implement the feature translation system, and the technology that we will use to implement the system. The research topic I found useful are listed in the references section. In A term we cemented the nature of the general system. We created the org.jprotocol project to be a system of translating application messages. Having a translation system allows the framework to send altered messages to the client, and would also allow for adding backward compatibility in with old clients.

Time would have been dedicated to programming, but we were faced with technical problems. Mainly the computer that I was relying on had to be serviced which kept me without a private computer until near the end of the term. At that time it was decided that project time would be devoted to implementing the general and Ezenia specific frameworks.

The general framework was designed and written in B term but once again we ran into technical problems. The laptop that was loaned to me by Ezenia for development died and I could not use it to work on Ezenia's code. Eventually we set up a VPN connection to develop off of a remote development environment, but at this point it was too late to do much more that familiarize myself with the code base. Because no progress had been made with the server code and because priorities were changing within Ezenia over whether Ezenia wanted to support an iPhone client, it was decided at this point to spend C term on the server implementation of Ezenia's codebase. This would allow Ezenia to expand into whatever other

platform they decided and would allow for the existing whiteboard system to add some backward compatibility.

In C term time was spent finishing the final MQP paper, adding a demonstration example to show off the org.jprotocol system, and fixing small defects found in the system during implementation. Although some technical problems did show up in C term, (inconsistent internet) they were not as serious as they were in earlier terms. However I was still unable to deliver a server implementation for Ezenia and the fault lies with me for that.

In retrospect I overestimated the amount of time I had available and overestimated the amount that I could deliver in that time. In addition to my failure to manage my time well, I needed to finish my minor capstone in C term along with the MQP. The end result was that my C term MQP time became devoted to improving the public side of the MQP. I hope to use my time in D-term to make up for this and finally deliver the server side implementation to Ezenia. However this would need to take place outside the scope of the MQP.

Benefits

The jprotocol extension is built to allow different versions of clients and servers to communicate easily with each other. The extension allows comparison, identification, and translation between different protocols. When presented with two protocols, for instance a client whiteboard protocol and a server database protocol, each side's protocol is capable of telling what messages can and can't be sent between the two. Protocols can perform *diff* operations between each other including: checking if both protocols are equal, checking which messages are shared between protocols, and checking which messages are missing from one

protocol but present in the other. The extension allows translation of messages to be performed on clients and servers, so that translation can be offloaded onto clients, lightening the workload on servers.

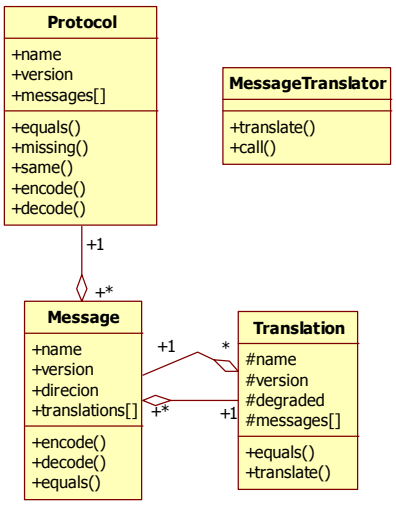
The main benefits of using the jprotocol translation system are:

1. Expansion. Different protocols across different platforms can interoperate easily. New platforms are easy to add because partially implemented protocols do not cause errors.

2. Backward Compatibility. Because different versions can now operate together on the same system, updates do not need to be forced between all clients, making it easier for customers.

3. Testing becomes easier. For the same reason that different platforms are easy to add, prototypes can be added to the network and can interact with previous, stable versions of clients without needing to change the clients.

Design



Protocol

The Protocol class represents the group of message types that a node, such as a client or a server, can use to communicate. The protocol class is uniquely identified by its name and version and contains a group of messages. The messages version state can be set to version number, version date, or whatever else the individual network may require. This is why both name and version are strings. The only real requirement is that all versions of a Message type be unique strings. Because the system was build to interact with Jabber, I use version number, since that is what Jabber xml uses to mark its versions.

There are three methods that protocols can use to compare themselves to each other. Protocols can do a binary check if the name and version are equals() each other. They can also do diff-like operations which create a new protocol of the operations that are shared between two protocols like so: A.same(B). Or a new protocol of the messages that are missing from one protocol but present in another like so: A.missing(B). This method only gives the message types missing from protocol B that are present in protocol A. To get the messages that are missing in A that are present in B, you need to check which messages are missing in B like so: B.missing(A). Protocols can be encoded into and decoded from serializable objects to facilitate network transmission. However, because the type of object is dependent on the specific network, this capability cannot be provided by the jprotocol framework but must be written by the user. For instance, when sending a class over xml you do not send the class itself, but the important data of the class as strings.

Message

The message class represents a specific type of message that can be sent over a network from one node to another. Messages are uniquely identified by their name and their versions. Messages may contain translations and contain a direction if the message can only be sent to certain types of nodes. For instance some messages are only sent to servers from clients, or vice versa.

Translation

The translation class represents a single translation from one message to one or more messages. Each translation is uniquely identified by its name and version. Each translation is marked as degraded or not and contains the messages that the original message will be translated into. Translations can check if they are equal to each other, and they contain the information needed to translate its intended message. Note that the translation does not translate the message *class* but the actual message.

Translations can encapsulate some complex and useful behavior. They can handle simple transitions from a new message type to an old message when dealing with older systems. They can also have utility when data needs to be altered, such as translating US measurements to the metric system. The more complex behaviors become apparent when translating a single Message multiple times and/or into multiple Messages.

For instance, if network speed was improved by combining multiple messages into a single message, the translator can still bridge the communication gap. Additionally, if there have been multiple iterations of a system (A, B, and then C for our purposes) then when A

communicated with C, A can translate it's Messages into B type messages and then translate the B type Messages to C type Messages. This allows you to stay compatible with multiple different older versions, without doing extra work.

MessageTranslator

The message translator class is the class that actually translates messages into other messages. It can give a list of the translations required to translate a message from one message into one or more other messages. If given a list of translations, the MessageTranslator can then perform them on a message and return a translation. The translation process is broken up into two steps because once the specific translation required has been found, it can be cached so that it can be applied multiple times without the expense of looking it up again. The MessageTranslator is currently set to find the optimal translation. For our purposes the optimal translation is degraded as little as possible and is broken down into as few new messages as possible. This optimization is a hard coded behavior for the MessageTranslator. However if you wish to have a Translator with different optimization goals, simply create a subclass of MessageTranslator with a different translate() method.

Approaches to managing Version Incompatibilities

As software systems evolve, developers assign versions to major "releases" of the software. Each version can be considered to support a number of *features*. A feature is an externally visible behavior of an application. Given a client/server system by which the client communicates to the server, one can detect new features simply by inspecting the communication protocol. The basic idea is to identify a mapping from software version to a set

of provided features. That is, each version can be said to definitively support (or not support) a specific feature. From this information, one can identify a "lifecycle" of sorts for each feature. Given a feature, one can detect the system version which supports the earliest manifestation of that feature. Typically features appear and then are supported for the duration of a software system's lifetime. Occasionally, a feature is "deprecated" which means it is no longer supported and the implication is that it won't be supported again in the future. In practice, however, most systems with deprecated features continue to use and even rely on those features indefinitely.

The first step to managing the incompatible communication of two software versions is to be able to detect when it takes place. This is done using one of four feature mapping models. If all features are additive, then more recent versions of an application support a superset of the features of the earlier version.

By Version Feature Discovery

In this option, each client has a version number, and from that the server can know what features the client has. The benefit of this is that it saves a lot of overhead in sending features and feature versions back and forth between the server and client, so it improves performance. Performance improvements come in the form of limiting needed bandwidth and the processing of individuals features and versions, imagine sending a feature listing for every single feature and version that the client supports every time you log in or start whiteboard and you'll have an idea of the overhead you may be inviting, especially if you consider new features added in every new release The downside of this option is that it requires synchronizing the code between the server and the client after each change, which slows down development and

makes it impossible to translate features if the version has not been transcribed, as well as being contrary to the idea of a feature discovery system. On the other hand, there will most likely be much fewer official versions shipped to customers than versions made during development, so you may want to include this for prominent release versions to of your clients and servers.

By Feature Discovery

In this option the client and the server list all the relevant features and versions that the each is willing to support at login time. This has a big advantage where the client may not be allowed to use some features, and the server can prevent the client from sending those Messages by withholding those Messages. It's also the default feature discovery method and probably the easiest to implement. Also this only entails a slowdown for the client at launch/login time. As long as their server remembers what the client's implemented features are, then it can interpolate between them easily. The Protocol class best supports this method and so code example uses this method.

Just in time feature Discovery

This is a more complicated but it may be worth using if you support a very large set of features, but use only a small subset of them. Rather than listing the features the server is using at runtime, just have the client list the version of the feature they are calling when they call it.

If the versions are the same, then no discovery methods ever have to be called, so this option only gets faster the more aggressively you have clients update. If the server sends an

incompatible feature to a client (for instance when you have two or more different client version active), that client can respond with its version number of the feature or that it does not have that feature. At that point the server can convert the sent feature to another feature, or send the client a message that it is not compatible. At this point the server can remember for the duration of the session (or even only for the duration of the whiteboard session for instance) how or if to translate the feature.

This means that because most of the features will be unused for each session, the client will be faster. The downside is that if the client is very old and all of its features are incompatible, you trade a slowdown at the client start, to a slowdown spread out during the run of the client.

Forced Update Discovery

Sometimes features cannot be translated. This means that required updates on the client may be a better solution than trying to force the server to a translation method. Realistically this will most likely be required for many situations, but mapping and substitution should be used to minimize the amount to updates required.

Existing Solutions

Extensible Messaging and Presence Protocol (XMPP), formerly known as the Jabber protocol, is the communications protocol used by Ezenia's MxM client as well as the Info Work Space (IWS) client, their primary product. The clients follow the XMPP [3] standard closely, but deviate to support extensions and features not available in other collaboration tools.

CVS is one of the most popular source control programs on the market, and being open source, I got a chance to look over its code. CVS is well known for its inter-version support [6]. The truth is that CVS uses all four methods of feature discovery. The main CVS client has well known versions, so when the server knows the version number of a client, the server knows what features the client supports, which ones it doesn't, and the ones that need to be substituted. But as an open source program it also supports feature listing from unfamiliar clients. Also because some features are not supported on all machines, for instance, some are banned by administrators. In addition some protocols, like the remote protocol, are only backwards compatible to a certain point before users are forced to upgrade to a newer CVS version.

CVS has a lot of backward compatibility, but only for a few versions back. Looking over their code I did not learn how they handle backward compatibility between features. I did learn that they had to leave a lot of small code blocks for backward compatibility. This doesn't seem like a great way to handle backward compatibility. Also while the Microsoft compatibility pack is advertised a lot, there is little to explain how it works. Since I am expecting to extend the service discovery extension of jabber, I will most likely base my implementation off that.

Jini is a network architecture for distributed systems [7]. Services are discovered and then leased to an object in the network for a set time. In other words, rather than translating between features to avoid problems with incompatibility, services can be guaranteed to be up to date, because you only connect to them for a certain time and then have to re connect and check that the service is up to date. Although this is not how we wish to create the feature

discovery system, this is a good example of a way to keep distributed systems constantly up to date at the same time.

A Service Oriented Architecture (SOA) is a system where features are packaged as interoperable services. Services are as loosely coupled as possible so that the services can be remixed and reused. Services are defined by what they do and how the communicate (i.e. functionality and protocol). In other words, in SOA each feature is its own module that can be plugged in to a larger system where each module knows how it communicates and what it does. The generalized feature discovery system could be adapted to be a discovery system for an SOA architecture, showing which interoperable services could be mapped on to others. However the whiteboard features are not organized to be like the SOA architecture. SOA architecture seems to apply best to enterprise level systems. Each interoperable service package must meet many requirements, so you trade a certain amount of overhead for the fact that the service will probably be reused many times.

The system we are building falls under the domain of Computer Supported Collaborative Work applications. In a Computer Supported Collaborative Work application, users use the tool to communicate and work together on shared information. CSCW refers not only to the applications, but also the study of how people use them. Applications can be broken down into four groups, based on whether users use it at the same time or different times and whether they are in the same place or in different places. For instance, the whiteboard MxM application is a same time/different place application. Because CSCW applications encompassed everything from massive multiplayer online games to email, this is a large are of study. [10] [11]

Translation Behaviors

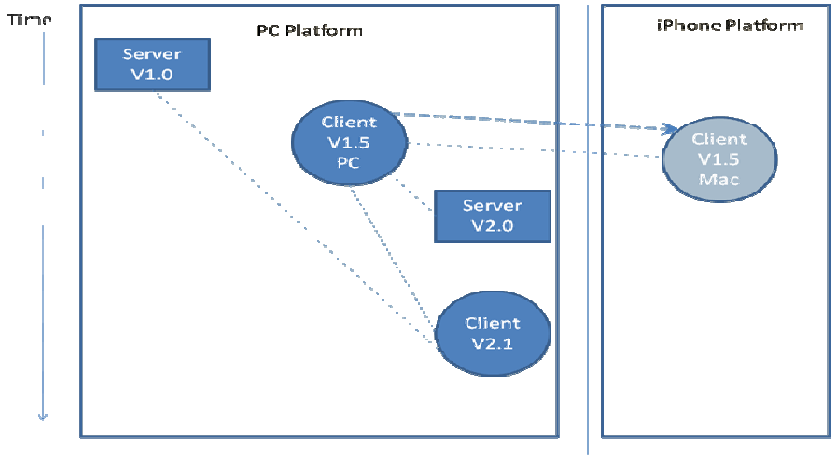
There are several translation behaviors that the server can use when translation is required. The first is the default behavior for dealing with unknown features, which is to ignore them. This is unacceptable for a feature translation service so the client we develop should do better.

A slightly better behavior is to have the server send a message to the client that the feature cannot be translated. This is a good fallback behavior if nothing better can be sent to the client because you can always do this, and the user can know that there is something that they don't see.

A better solution than that is to send a substitution of a different but similar feature or an older version of the same feature. For instance, a low quality .jpg image file can be sent when a high quality .png image file cannot be sent. Obviously the client must be notified that there was a substitution. This is a good solution and is a better choice than just sending a message of incompatibility, but we can still do better in many situations.

Even if a certain feature is not available it can often be mapped into another feature. For instance, if one client tells another to draw a box, and the other client does not know how to draw boxes, but does know how to draw lines, then the server can translate the box message into four line messages. This is the optimal solution when feature translation is required. Also note that mapping can work recursively, so that if boxes can be mapped to lines and lines can be mapped to points then boxes can be translated into points. Further, this means that taking advantage of this translation mapping hierarchy can allow a client to display many features while only implementing a small amount of them.

Design Challenges

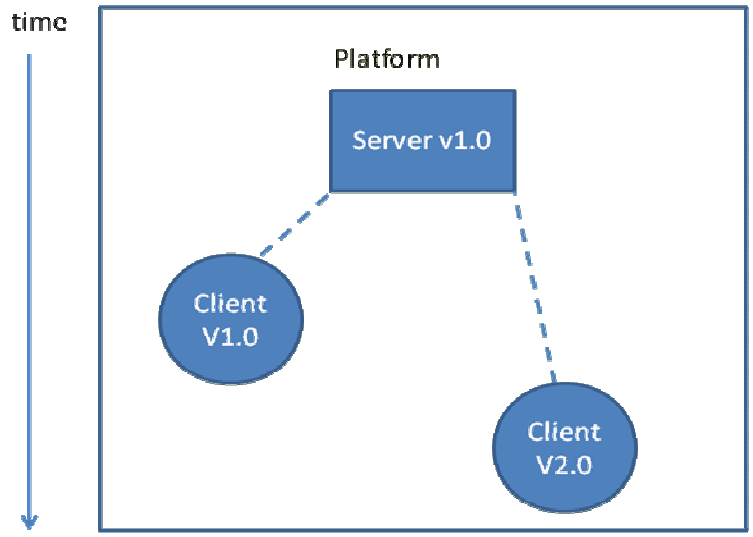


There are five main incompatibility problems that that the proposed framework will have to manage. In order to more simply demonstrate how the message translation system works I am going to refer to a real life example: translating spoken languages. As anyone who has used Babelfish or Google Translator can tell you, translating languages is hard for a computer because of the complex rules involved in human speech. (Google Translator and Babelfish are web services that act as human language translators. They are run by Google and Yahoo respectively. They can also be accessed programmatically to translate websites or as translators inside a program.)[12][13]

However we only need to translate one message at a time, the equivalent of translating one word at a time, which is something even language translation programs can do well. This is because translation currently only goes in one direction (new->old) to prevent infinite loops. This means that the system could say convert a square Message into four line messages, but the four line messages will not be translated back to a square, they would only appear so on the whiteboard screen.

As I already mentioned, Ezenia uses client/server architecture as opposed to peer to peer architecture. Peer to peer architecture is the equivalent of two people talking face to face; the two programs are peers. Client/server architecture is a closer equivalent to a speech. In a room full of people (clients) there is one speaker (server) who takes questions from a person in the crowd and then relays them to everyone in the crowd. Rather than sending messages directly to each other, messages are filtered through a server and sent to the relevant clients. However, as you will see, the jprotocol system can deal with both types of architectures.

Here are the main design challenges we need to overcome when translating messages.



The first problem design challenge is translating features between two clients where one client is an earlier version than another. Because the target application is a Computer Supported Collaborative Work (CSCW) application, it is essential for clients to be able to communicate with each other “through the server”, which is an additional constraint not typically found in Client/Server applications where the clients only communicate with the Server. This can be easily solved by the translation system.

Imaging that an English speaking man from the 16th century has traveled to the future and is trying to speak to a modern day English speaker. (That's quite a version difference.)

```
Protocol WilliamShakespeare = new Protocol("English", "1600.0",
                                           shakesperianEnglish);
Protocol GeorgeBush = new Protocol("English", "2010.0",
                                           modernEnglish);
```

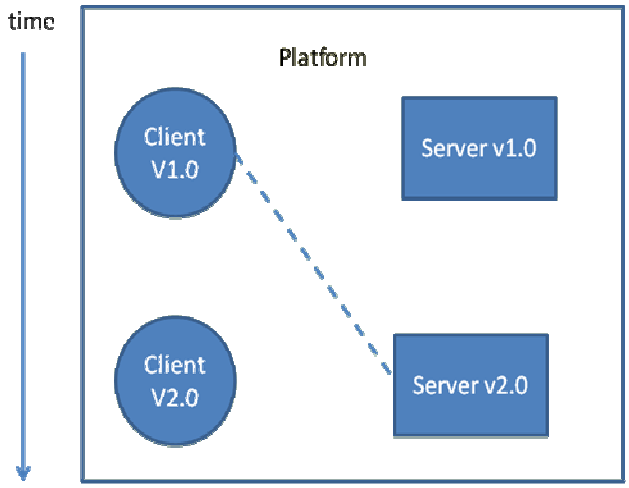
You might imagine that these two men would have trouble communicating. Luckily they have hired an interpreter (the server) who knows modern English and old English.

```
Protocol ShakespeareScholar = new Protocol("English", "2010.0",
                                           modernAndOldEnglish);
```

Now when the two need to talk to each other the server can translate smoothly and the two men can talk.

```
Translation1=translator.translate(new 16CenturyGreeting(),WilliamShakespeare,
ShakespeareScholar);
AssertTrue(translations.size==1);
Translation2=translator.translate(new 21CenturyGreeting(),ShakespeareScholar,
GeorgeBush);
AssertTrue(translation2==NULL);//no translation needed
String greeting= translator.call(translation1,"greetings");
AssertTrue(greeting.equals("hi"));
```

Notice that the clients and the server simply use the same class, protocol. This is why we can use the jprotocol system for peer to peer or client to server systems, because the translator doesn't care if the other protocol is a client or a server. Communication becomes symmetrical.

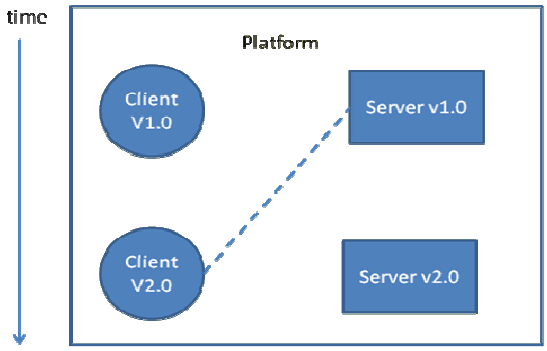


The second design challenge is translating features from updated servers to outdated clients. Older Client applications should still be able to receive at least a minimal degraded form of communication when the base Server has upgraded. To enable this capability we will investigate how to enable the framework to support the automatic translation of "newer" communication to "older" means known by the older Client. This is also easily handled.

Let's go back to our example of the 16th century English speaker, 21st century English speaker, and 21st century translator representing an old client, new client, and new server. We've already seen that the server and the old client can communicate a greeting. But what would happen if they encountered a message that could not be translated?

```
translations=translator.translate(new strangeOldWord(),WilliamShakespeare,
ShakespeareScholar);
assertTrue(translations.iterator().next().equals(new FailureTranslation()));
```

Notice how, even if the message were to fail to translate, we could check that rather than causing a crash when the message was delivered. The client can simply generate a message that translation is not possible.



The third is translating features between updated clients and outdated servers. Traffic must go through and be translated by a server so clients must be able to deal with outdated servers and servers must be able to deal with outdated clients.

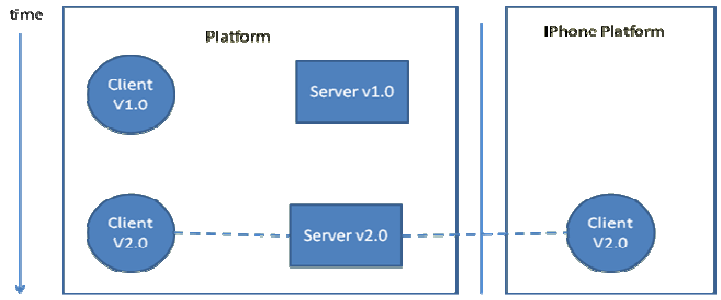
In order for newer clients to work with old servers, the new clients must remember the commands that were available in old servers or else they will fail. If we have a 16th century scholar step in as the old server, he still wouldn't be able to talk to a modern English speaker, because he would have never heard modern English.

```
translations=translator.translate(new 16CenturyGreeting(),GeorgeBush,
                                          16thcenturyTranslator);
assertTrue(translations.iterator().next().equals(new FailureTranslation()));
```

But if the 21st century speaker were to remember some of the old language (backwards compatibility) then he would be able speak to the translator and communication can resume.

```
GeorgeBush.messages.add(new greetings());
```

```
Translation1=translator.translate(new 16CenturyGreeting(),GeorgeBush,
                                  16thcenturyTranslator);
assertTrue(translation1==NULL);
translation2=translator.translate(new 16CenturyGreeting(),
                                  16thcenturyTranslator, WilliamShakespeare);
assertTrue(translation2==NULL);
```

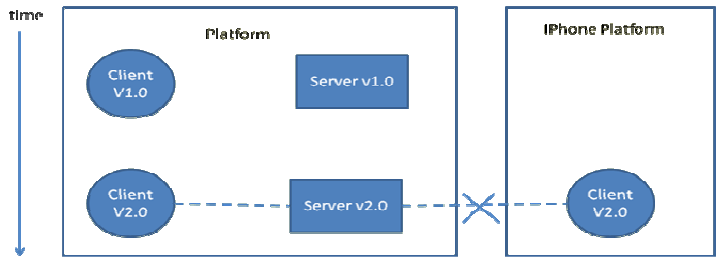


The fourth problem we must deal with is feature translation on clients on different platforms which are not able to display the same things. Often porting software from one platform to another requires some features to be deprecated (i.e., “be careful when using”) or even eliminated. Because the focus of the project is developing a whiteboard project, most of the project will focus on translation of features between clients.

Here we refer to a chat between two clients on different platforms. An equivalent to this would be a chat between an English speaker (client) a French speaker (client) and an interpreter (server). In order for the two clients to speak, you need a server that can speak with both clients.

```
Protocol Englishman = new Protocol("English", "2010.0",
                                                modernEnglish);
Protocol Frenchman = new Protocol("French", "2010.0",
```

```
                                                    modernFrench);
Protocol Translator = new Protocol("client", "2010.0",
                                                    new ArrayList<Message>());
Translator.messages.add(modernEnglish);
Translator.messages.add(modernFrench);
english2frenchgreeting =translator.translate(new 21CenturyGreeting(),
                                            Translator, modernFrench);
String result =(String) translator.call(english2frenchgreeting, "hi!");
assertTrue(result.equals("salut!"));
```



The fifth problem is that some features may be completely incompatible with the platform that a client is running on. Because this project involves supporting a whiteboard client on limited devices, such as a mobile phone, it is likely that there may be some features that will not run and the framework will plan for such an event and enable the designers to provide translators as needed.

When translating English to French, some words cannot be translated because they do not exist in both languages just as some messages are not present in both clients. This means that they cannot communicate.

```
Protocol boardAMinusB= Englishman.missing(Frenchman);
assertTrue(boardAMinusB.messages.size()==1);
```

At least not directly. The good news is that the translation system can address this. Instead of directly translating a word that has no translation (say “tentacle”) you can translate a similar word that does exist in both languages (say “limb” translated to “membre”). In the same way you can translate messages to other similar messages in order to get across some message to the other client. This is the purpose of the depreciated translations.

```
translations =translator.translate(new octapusLimbWord(),
                                   Translator, modernFrench);
String result = (String) translator.call(translations, "tentacle");
assertTrue(result.equals("membre"));
```

Note here that even though the translator used a depreciated translation, it would not do so if there was a better choice. Also note that in all of these challenges, the steps to solve them have been identical. This makes the translator much simpler to use.

Design Context

All of this work will be accomplished within a client/server architecture. At the same time, this is a Computer-Supported Collaborative-Work (CSCW) application [9], which typically implies greater interactivity among the participating clients. MxM is an application that is designed to allow people to communicate with each other, so the end goal is to always allow users to communicate with each other.

Ezenia designs communication systems for the military, so certain data and features are purposely not available to all users. There are additional design concerns because of the

military domain. For instance users should not be able to exploit the feature translation system to send hidden messages.

In MxM, all communication takes place in rooms. A client sends a message to the server and the server passes that message on the other clients in the room. This behavior works the same whether the client is sending a simple text message or sending an image in the whiteboard. We will take advantage of the room context when supporting feature translation system on the server side. The translation system simply translates the message for each individual client, and sends the appropriate message to each one. Thus we will be able to centralize the translation code on the base Server system.

How it works

When the client connects to the server and authenticates, the client and server will need to know what protocols the others support. At this point the client and server can encode() the protocol in question and exchange protocols over the network.

```
ProtocolSerial serialProtocolObject =((EncodingProtocol)boardProtocol).encode();
```

After decode()ing the other's protocol, the client and server can run a missing() method, to see what commands they cannot send to each other.

```
Protocol boardAMinusB=clientProtocolA.missing(ServerProtocolB);
```

As in this example, at this point the server and the client know the protocol they have, the protocol of the entity they are trying to communicate with, and the difference between the protocols. At this point the two entities know enough to send each other messages in this

protocol. Depending on the implementation, the entity can either keep a hash table of how to translate each message to the other entity or it can just figure it out each time it wants to send a message. Calling MessageTranslatior.translate() returns a list of translations that will need to be applied to the message you want to send.

```
MessageTranslator translator =new MessageTranslator(new FailureTranslation());
Collection<Translation>translations=translator.translate(new TextMessage(), from, to);
```

Call()ing the list of translations on the intended message transforms the message into another message or group of messages that other entity will be guaranteed to understand. (Even if that message happens to be that the message can't be translated.)

```
String result =(String) translator.call(translations, "some text");
assertTrue(result.equals("c:/some text.jpg"));
```

Implementing the system

In this section I use examples from the public org.jprotocol example project and code testing to demonstrate my points.

The MessageTranslator, Protocol, and Message classes can be created directly by using their built in constructors.

```
Protocol boardProtocol9=
      new Protocol("imageboard", "0.9", new ArrayList<Message>());
DemoMessage demo =New Message("demo","1.0",
      Message.Direction.clientToServer, "hello",null);
MessageTranslator translator =
```

```
        new MessageTranslator(new FailureTranslation());
```

However in order to encode properly for your network, you will need to subclass the protocol and message classes.

```
public static DemoProtocol recognizedMessages =
        new DemoProtocol("main", "1.0", null);


public DemoMessage(String message, String sentFrom) {
        super("demo", "1.0", Message.Direction.clientToServer,
                message,new ArrayList<Translation>());
        origin=sentFrom;
}
```

More importantly the translation class needs to be subclassed to create each instance of a translation from one message to another. For this reason it is more convenient to have a Message subclass for each type of message. That way you do not need to keep adding the correct translations after each message gets instantiated.

```
public GetMessage(String message, String sentFrom) {
        super(message, sentFrom);//superclass = DemoMessage
        name="get";
        Message mess=new demo.old.messages.GetMessage(message, sentFrom);
        this.translations.add(new NewToOldTranslation(mess));
}
```

Keep in mind when creating Translation classes, translation happens from new to old. Making a translation from old to new and another for new to old is both unnecessary for translation to work and may cause infinite loops.

```
public TossToDropTranslation()
{
	//The subclass decides these values.  They will always have the same
default when the class is new.
	name="TossToDrop";
	version="1.0";
	degraded=false;
	messages=new LinkedList<Message>();
	messages.add(new DropMessage("?", "?"));
}
```

About The Demo

The demo is an example to demonstrate feature discovery, feature translation, and backward compatibility translations. The actual example is a many to one client to server example. Messages sent to the server by one client are passed to the other clients connected to the server. The program emulates an old school text adventure game. The server stores the state of the items, while clients can send messages in the form of <verb> <item> (for instance: open book). Currently users can: get, open, close, drop or toss items, or take an inventory of the items on the server (inv all or inv user).

```
inv all
Client received:demoMessage [Received on server:invMessage [all]]
Client received:demoMessage [Items on server: ]
Client received:demoMessage [Name   Available   Open  Owner]
Client received:demoMessage [book   true        true  none]
Client received:demoMessage [bomb   true        false none]
Client received:demoMessage [candy  false       true  user76]
```

*Basic translation is demonstrated with the 'old client' where the server cannot receive a toss message, so the client sends a drop message instead.

```
toss baseball

Client received:demoMessage [Received on server:dropMessage [baseball]]
```

The translator knows to do this simply by adding the correct translation.

```
public TossMessage(String message,String theOwner) {

	super(message, theOwner);

	name="toss";

	this.translations.add(new TossToDropTranslation());

}
```

The translator also helps the client to fail gracefully without crashing, by sending a failure message when a message cannot be translated.

```
read book

Client received:demoMessage [Received on server:

	failureMessage [cannot translate message: read book]]
```

*Backward compatibility translation is demonstrated by having the new and old clients and servers communicate. The newer client and newer server are both capable of translating their newer messages into older messages for communication. I marked the newer messages as new so they would appear so in the text output. Note how the chat differs based on the server.

On an old server all chat reverts to the level of the server.

On the new client

```
open book
Client received:demoMessage [Received on server:openMessage [book]]
Client received:demoMessage [Received on server:closeMessage [book]]
```

On the old client

```
Client received:demoMessage [Received on server:openMessage [book]]
close book
Client received:demoMessage [Received on server:closeMessage [book]]
```

However communication on newer servers is more interesting.

On the new client

```
close book
Client received:demoMessage [Received on server:closeMessage
[book]*new*]*new*
Client received:demoMessage [Received on server:closeMessage [box]]*new*
```

On the old client

```
Client received:demoMessage [Received on server:closeMessage [book]*new*]
close box
Client received:demoMessage [Received on server:closeMessage [box]]
```

The server sends and receives old messages from old clients and new Messages to new clients. Note that here the receiving clients only get a string informing them of what happened on the server wrapped inside a message. If we actually passed on a each message from the server to all the clients then only messages sent from an new client to a new server to a new client. If

any oldMessage is translated to somewhere in the communication chain, then an oldMessage will be the message that arrives at the destination client.

*feature discovery is demonstrated in the demo by the loginMessage class. When a client logs in, the client sends a login message that includes its protocol, and the server does the same.

```
public class LoginMessage extends Message {
	public Collection<Message> messages=new ArrayList<Message>();
	public LoginMessage(String sentFrom,DemoProtocol prot) {
		super("login", "1.0", null,sentFrom, null);
```

They both then run a setup method, which translates each of its messages and then stores the translation in a hashtable the translation.

```
MessageTranslator translator = new MessageTranslator(new
	FailureTranslation());
static Hashtable<String, Collection<Translation>> translateTable =
	new Hashtable<String, Collection<Translation>>();


public void setup(Protocol fromMessages, Protocol toMessages) {
	Collection<Translation> empty=new LinkedList<Translation>();
	Iterator<Message> messages=fromMessages.messages.iterator();
	while(messages.hasNext())
	{
		Message mess= messages.next();
		Collection<Translation> trans=translator.translate(mess,
			fromMessages, toMessages);
		if(trans!=null)
		{
```

```
                    translateTable.put(mess.name+mess.version, trans);
            }
            else
            {
                    translateTable.put(mess.name+mess.version, empty);
            }
      }

}
```

This one time translation means that the messages that can be sent have been discovered. This upfront cost reduces the dynamic cost of communicating, because now the translation only needs to be applied before sending the message.

```
public Message translate(Message in) {
      Message out=(Message) translator.call(
                              translateTable.get(in.name+in.version),in);
      return out;
```

Evaluation

There are two main avenues that need to be evaluated, the public academic portion of the project and the non public portion to be delivered to Ezenia.

The public system was to be designed so that it could be used to implement the private version of the translation system. In this way the public system was more important. Additionally, it was difficult to analyze how much could be delivered to Ezenia, because of the proprietary nature of their system.

The academic portion of the MQP required three things: a backbone translation system that could be implemented on Ezenia's server or any other network, a publicly displayable example that the translation system works, and the papers required for an MQP.

The Ezenia portion of the MQP is, as mentioned, harder to pin down. While I am discussing with Ezenia delivering some of the MQP in D term, that would be outside the scope of the MQP. At this point it is up to Ezenia to judge whether the academic portion being delivered is sufficient.

In the academic portion of the MQP, the main translation the feature system must be able to:

- Map features and different version of features to each other.
- Perform mapping recursively.
- Perform a substitution if there is no mapping available.
- Indicate to the client when a feature cannot be translated all.

The clients should be able to identify what features they have by either giving a client version number or giving the version number and name of each relevant feature. Just as important there must be an easy to write pattern or system to implement mapping between features. If programmers don't take the time to write mapping code then the feature translation system is literally useless. Therefore the mapping code should be as simple and quick to implement as possible. I believe that I have succeeded in the obvious requirements. The ease of use requirements will need to be decided upon by Professor Heinemann.

Originally, the translation system was supposed to be deployed on Ezenia's servers, and then a public version of the translator would be released as an example, with code altered so that no confidential information from Ezenia could be given away. However without delivering anything on Ezenia's servers, a public example became more important.

The public example was intended to be small, but still show how the translation system could be useful on a real world system. The public example was to be a small server/client chat reminiscent of old text adventure games. Users could get, drop, open, and close items. We intentionally added an incomplete toss method to demonstrate incompatibility and translation. We also play to have two different versions of the client and server to demonstrate backward compatibility. The effectiveness of the example will be judged by professor Heinemann.

Finally I must be evaluated by Professor Heineman on the final MQP paper that I deliver and on my ability to manage the project over the course of the three terms. This will probably have the most weight on my grade, because my performance as a student can be measured here independently of any setbacks.

Evaluating the Client

The effectiveness of the client is based off of how easily it could be adapted for use in another system. In order to have a working translation system implemented on another network, several things will be needed.

*The org.jprotocol framework

The generic org.jprotocol framework that I use as a base will need to be included. These files (found in the shared folder) can be included without changes.

*A Feature Discovery System

The client and server both need to be able to tell what messages the other is capable of sending. This means that a user either has to write their own feature discovery system (they may want to use a different type of feature discovery), or use methods from the demo.

A simple feature discovery system can be assembled from a MessageTranslator, a hashtable, and two of the methods found in CommunicationAgent.java.

```
MessageTranslator translator=new MessageTranslator(new FailureTranslation());
static Hashtable<String, Collection<Translation>> translateTable =
      new Hashtable<String, Collection<Translation>>();
```

The setup method hashes the translations so that they can be called later.

```
public void setup(Protocol fromMessages, Protocol toMessages) {
      Collection<Translation> empty=new LinkedList<Translation>();
      Iterator<Message> messages=fromMessages.messages.iterator();
      while(messages.hasNext())
      {
            Message mess= messages.next();
            Collection<Translation> trans=translator.translate(mess,
fromMessages, toMessages);
            if(trans!=null)
            {
                  translateTable.put(mess.name+mess.version, trans);
```

```
            }
            else
            {
                  translateTable.put(mess.name+mess.version, empty);
            }
      }
}
public Message translate(Message in) {
return (Message)translator.call(translateTable.get(in.name+in.version),in);
}
```

The system also has to be able to send the servers or clients protocol over the network. You could use the LoginMessage class to send it:

```
public class LoginMessage extends Message {
      public Collection<Message> messages=new ArrayList<Message>();
      public LoginMessage(String sentFrom,DemoProtocol prot) {
      //cut off
```

At this point very little code needs to be written, but the network needs to be able to read the code.

*Adapt the system to their network

This was the most time consuming step of the demo, but this setup step is only a one time cost, and the time commitment will vary from network to network. Basically, the system needs to be able to send messages over the intended network. The framework comes with methods that can be implemented to encode Messages, Protocols, and Translations.

```
SerialObject serialProtocolObject =exampleProtocol.encode(); //example
```

Alternatively, if your network is capable of sending Message objects directly you may need to sanitize them to function correctly on your network. For instance, the CommunicationAgent class in the demo was not capable of sending classes over the network if they grew too large, so methods were added to strip Messages of non-essential data before sending them over the network. This was probably a bug on the part of CommunicationAgent, but it still needed to be adapted to.

```
Message msg=DemoProtocol.encodeMessage((DemoMessage)msg);
return agent.writeObject (msg);
```

If you look closely in the demo you will notice many small alterations I made to avoid bugs on the CommunicationAgent. Unfortunately, because each network is different this will need to happen individually.

*Message types

Finally, in order translate between different messages, the system needs to have Message classes and Translation classes written. In the case of the demo, I uses a Message class for each different kind of verb. In a real system, you may want to use a Message class for each type of network Message. (For instance on a whiteboard system you would need a line class, a shape class, an image class, etc.)

```
exProtocol.messages.add(new demo.old.messages.TossMessage("?",name));
exProtocol.messages.add(new demo.old.messages.DropMessage("?",name));
exProtocol.messages.add(new demo.old.messages.GetMessage("?",name));
exProtocol.messages.add(new demo.old.messages.OpenMessage("?",name));
```

```
exProtocol.messages.add(new demo.old.messages.CloseMessage("?",name));
exProtocol.messages.add(new demo.old.messages.InventoryMessage("?",name));
exProtocol.messages.add(new demo.old.messages.DemoMessage("?",name));
```

For each Message you may want to translate into another Message you will need to add a Translation class to that Message class.

```
public GetMessage(String message,String sentFrom) {
	super(message, sentFrom);
	name="get";
	Message mess=new demo.old.messages.GetMessage(message, sentFrom);
	this.translations.add(new NewToOldTranslation(mess));
}
```

The bad news is that for each system, you need to write your own subclasses and translations for each message type. The good news is that they are usually written very quickly and because the Messages, Protocols, and Translations themselves are independent from the network, so they can be ported easily.

Conclusion and Future Work

The feature translation system will be an asset to Ezenia as the company expands its software into new areas and platforms. The feature translation system allows the system to stay flexible, while still allowing for legacy support. This will minimize the amount of time spent on maintenance and make the application easier to expand.

This system is open to expansion into other works. If the theoretical framework is sound, it could be applied to any system that needs network features translated as the system evolves.

Further work could be to develop a feature translation system to interpret between different but related clients. For instance, Ezenia is looking to make their different product lines to communicate with each other and other translation enabled systems.

References

1. Apple, iPhone Developer Site, http://developer.apple.com/iphone/, Retrieved Feb 26, 2010

2. Ezenia Corporation, Corporate Website, http://www.ezenia.com/, Retrieved Feb 26, 2010

3. XMPP Standards Foundation, http://xmpp.org/, Retrieved Feb 26, 2010

4. Service Discover in Jabber, http://xmpp.org/extensions/xep-0030.html, Retrieved Feb 26, 2010

5. Scalable Vector Graphics (SVG) http://www.w3.org/Graphics/SVG/, Retrieved Feb 26, 2010

6. CVS source Control, http://ximbiot.com/cvs/, Retrieved Feb 26, 2010

7. Jini Technology, http://www.jini.org/wiki/Main_Page, Retrieved Feb 26, 2010

8. Xml.com, Service Oriented Architecture, http://www.xml.com/pub/a/ws/2003/09/30/soa.html, Retrieved Feb 26, 2010

9. Wikipedia, Service Oriented Architecture, http://en.wikipedia.org/wiki/Service-oriented_architecture, Retrieved Feb 26, 2010

10. John H. Saunders, A Manager's Guide to Computer Supported Collaborative, http://www.johnsaunders.com/papers/cscw.htm, Retrieved Feb 26, 2010

11. Wikipedia, Computer-Supported Cooperative Work, http://en.wikipedia.org/wiki/Computer_supported_cooperative_work, Retrieved Feb 26, 2010

12. Yahoo.com, Babelfish translator, http://babelfish.yahoo.com/, Retrieved Feb 26, 2010

13. Google.com, Google Translate, http://translate.google.com/Retrieved Feb 26, 2010